Mainframe Interview Preparation Guide

COBOL • JCL • DB2/SQL • CICS • Mainframe Tools

Consolidated Q&A Reference — Compiled from Accenture, Capita India, Deutsche Bank, Broadridge, CGI, IBM, and Deloitte interview rounds.

Table of Contents

1. COBOL — String Handling & Data Definition

Q1. How do you split a full name (e.g., 'RAJESH KUMAR KHANNA') into separate fields?

Answer: Use the UNSTRING statement, which breaks a single field into multiple receiving fields based on a delimiter (here, a space). UNSTRING is the standard COBOL tool for parsing variable-format text into discrete fields, and it also supports a COUNT IN and TALLYING IN clause to track how many characters were moved.

```
01 WS-A PIC X(100) VALUE 'RAJESH KUMAR KHANNA'.
01 WS-FIRST-NAME PIC X(20).
01 WS-MIDDLE-NAME PIC X(20).
01 WS-LAST-NAME PIC X(20).

UNSTRING WS-A DELIMITED BY SPACE
    INTO WS-FIRST-NAME, WS-MIDDLE-NAME, WS-LAST-NAME
END-UNSTRING.
```

Q2. How do you check whether a specific word (e.g., 'KUMAR') exists inside a string?

Answer: Use INSPECT ... TALLYING FOR ALL to count occurrences of the search value within the field. If the resulting counter is greater than zero, the value is present. This avoids manual character-by-character looping.

```
01 WS-NAME PIC X(20) VALUE 'RAJESH KUMAR KHANNA'.
01 WS-CTR  PIC 9(02) VALUE ZERO.

INSPECT WS-NAME TALLYING WS-CTR FOR ALL 'KUMAR'
IF WS-CTR > 0
    DISPLAY 'KUMAR IS FOUND'
END-IF.
```

Q3. How do you find the actual (non-space) length of data stored in a fixed-length field?

Answer: Loop backward from the last byte of the field using reference modification, counting characters until you hit a non-space byte, or use the intrinsic function FUNCTION TRIM combined with FUNCTION LENGTH for a one-line solution in modern COBOL compilers.

```
01 WS-CUST-NAME PIC X(20) VALUE 'RAJESH KUMAR KHANNA'.
01 WS-SUB  PIC 9(02).
01 WS-CTR  PIC 9(02) VALUE ZERO.

MOVE 20 TO WS-SUB
PERFORM UNTIL WS-SUB < 1
    IF WS-CUST-NAME(WS-SUB:1) NOT = SPACE
        ADD 1 TO WS-CTR
    END-IF
    SUBTRACT 1 FROM WS-SUB
END-PERFORM
DISPLAY 'NON-SPACE CHARACTER COUNT: ' WS-CTR.
```

Q4. How do you find the position of a specific character inside a field?

Answer: Scan the field one byte at a time using reference modification (FIELD(index:1)) inside a PERFORM VARYING loop, and display or store the index whenever a match is found. FUNCTION INDEX or FUNCTION INSPECT-based logic can be used as a shortcut in newer COBOL versions.

```
01 WS-NAME PIC X(20) VALUE 'XXXXXXXXXX'.
01 WS-SUB  PIC 9(02).

PERFORM VARYING WS-SUB FROM 1 BY 1 UNTIL WS-SUB > 20
    IF WS-NAME(WS-SUB:1) = 'X'
        DISPLAY 'POSITION OF X: ' WS-SUB
    END-IF
END-PERFORM.
```

Q5. How do you remove leading and trailing spaces from a field?

Answer: Use INSPECT TALLYING FOR LEADING/TRAILING SPACES with reference modification, or — far simpler in COBOL 2002+ compilers — use the intrinsic FUNCTION TRIM, which removes both leading and trailing spaces in one statement.

```
* Manual approach
INSPECT WS-NAME TALLYING WS-COUNT FOR LEADING SPACES.
MOVE WS-NAME(WS-COUNT + 1:) TO WS-TRIMMED.
INSPECT WS-TRIMMED TALLYING WS-COUNT FOR TRAILING SPACES.
MOVE WS-TRIMMED(1:LENGTH OF WS-TRIMMED - WS-COUNT) TO WS-TRIMMED.

* Modern shortcut
MOVE FUNCTION TRIM(WS-NAME BOTH) TO WS-TRIMMED.
```

Q6. What is the difference between leading spaces and trailing spaces?

Answer: Leading spaces are blank characters that appear before the actual data in a field (e.g., ' ABC'); trailing spaces appear after the data (e.g., 'ABC '). The distinction matters for comparisons, sorting, and when trimming fields before concatenation, since COBOL space-pads fixed-length alphanumeric fields by default.

Q7. What is the difference between PIC 9.99 and PIC 9V99?

Answer: PIC 9.99 stores an actual decimal point as a character, so it occupies one extra byte of storage (4 bytes for 3 digits + the period) and the decimal point is physically visible in the data. PIC 9V99 uses an implied (assumed) decimal point — V is a non-storage symbol — so it only takes 3 bytes; the value 123 is interpreted as 1.23 by the compiler but no period is actually stored. V is preferred for true binary/packed numeric storage since it saves space.

Q8. An arithmetic result overflows its receiving field (e.g., a 3-digit result moved into PIC 99) — how do you detect and handle this?

Answer: Use the ON SIZE ERROR phrase on arithmetic statements (ADD, SUBTRACT, MULTIPLY, DIVIDE, COMPUTE) to trap truncation/overflow before it silently corrupts data. Without it, COBOL truncates the high-order digits and keeps only the low-order digits that fit, producing wrong results with no warning. ON SIZE ERROR is paired with DIVIDE specifically to also trap division-by-zero scenarios.

```
01 WS-RESULT PIC 99.

COMPUTE WS-RESULT = 123
    ON SIZE ERROR
        DISPLAY 'OVERFLOW: VALUE TRUNCATED'
END-COMPUTE.

DIVIDE NUMERATOR BY DENOMINATOR GIVING WS-RESULT
    ON SIZE ERROR
        DISPLAY 'DIVISION OVERFLOW OR DIVIDE BY ZERO'
END-DIVIDE.
```

Q9. How do you normally define a counter field in Working-Storage?

Answer: Declare a numeric PIC clause sized for the maximum expected count and initialize it to zero, so it is ready to be incremented with ADD statements throughout the program. Using an unsigned numeric (PIC 9) rather than alphanumeric avoids implicit data-type conversion during arithmetic.

```
01 WS-COUNTER PIC 9(02) VALUE ZERO.
```

Q10. COBOL has no native VARCHAR — how do you represent a variable-length character field?

Answer: Model it as a group item containing two elementary fields: a fixed-length text buffer and a binary length field that records how many of those bytes are actually meaningful data. Programs that read the field use the length sub-field to know how much of the text buffer to process, mimicking SQL VARCHAR semantics used by DB2 host variables.

```
01 WS-VAR-CHAR.
   05 WS-VAR-CHAR-TXT  PIC X(04).
   05 WS-VAR-CHAR-LEN  PIC S9(04) COMP.
```

Q11. What is the length of a DB2 TIMESTAMP field when moved into a COBOL host variable?

Answer: A standard DB2 TIMESTAMP (YYYY-MM-DD-HH.MM.SS.NNNNNN) requires 26 bytes when represented as a character host variable, so it is typically declared as PIC X(26).

```
01 WS-TIME-STAMP PIC X(26) VALUE SPACES.
```

Q12. A DB2 table column is defined as DECIMAL(7,4) — what is the corresponding COBOL host variable?

Answer: Use a packed-decimal (COMP-3) field with the same precision and scale as the DB2 column: 3 integer digits and 4 fractional digits, expressed with PIC S9(3)V9(4) COMP-3. Matching precision/scale exactly avoids SQLCODE -301/-302 (invalid host variable) and silent truncation.

```
01 WS-VARI.
   05 WS-X PIC S9(3)V9(4) COMP-3.
```

Q13. What is the difference between COMP, COMP-3, and DISPLAY usage, and which is best for storage efficiency?

Answer: DISPLAY stores each digit as one full byte (zoned decimal) and is the easiest to read but the least space-efficient. COMP-3 (packed decimal) stores two digits per byte plus a half-byte sign nibble, roughly halving storage versus DISPLAY, and is the most common choice for numeric fields used in arithmetic. COMP (binary) stores the value as true binary and is the fastest for arithmetic operations and array subscripting, but is not human-readable in a dump. For storage efficiency, COMP-3 or COMP are both better than DISPLAY; COMP is generally fastest for CPU-bound calculations, while COMP-3 is the conventional choice for monetary/decimal fields exchanged with DB2.

Q14. What does the EDITOR (editing) PICTURE clause do in COBOL?

Answer: Editing PICTURE symbols (such as Z, $, comma, decimal point, CR, DB, +, -, and *) reformat a numeric value for human-readable display — suppressing leading zeros, inserting currency symbols, commas, and sign indicators. An edited field cannot be used directly in arithmetic; it is a 'report' version of a numeric field, typically populated by moving a numeric WORKING-STORAGE field into it just before printing or displaying.

```
01 WS-AMOUNT        PIC S9(7)V99 COMP-3.
01 WS-AMOUNT-EDIT   PIC $ZZZ,ZZZ,ZZ9.99CR.

MOVE WS-AMOUNT TO WS-AMOUNT-EDIT.
DISPLAY WS-AMOUNT-EDIT.
```

Q15. How do you concatenate several variables into a single field?

Answer: Use the STRING statement, which appends multiple source fields into one receiving field, optionally inserting delimiters between them, and tracking the ending position with a POINTER clause.

```
01 WS-PART1 PIC X(10) VALUE 'RAJESH'.
01 WS-PART2 PIC X(10) VALUE 'KUMAR'.
01 WS-PART3 PIC X(10) VALUE 'KHANNA'.
01 WS-FULL  PIC X(40).
01 WS-PTR   PIC 9(04) VALUE 1.

STRING WS-PART1 DELIMITED BY SPACE
       ' ' DELIMITED BY SIZE
       WS-PART2 DELIMITED BY SPACE
       ' ' DELIMITED BY SIZE
       WS-PART3 DELIMITED BY SPACE
       INTO WS-FULL
       WITH POINTER WS-PTR
END-STRING.
```

2. COBOL — Control Flow, Tables & Subprograms

Q16. What is the difference between PERFORM-PARA and GO TO-PARA, and which is preferred?

Answer: PERFORM transfers control to a paragraph (or range of paragraphs) and automatically returns to the next statement after the PERFORM once the paragraph completes, supporting structured, modular

programming. GO TO transfers control permanently with no automatic return, which can produce unstructured 'spaghetti code' that is harder to maintain and debug. Modern COBOL standards favor PERFORM almost exclusively; GO TO is mostly reserved for tightly controlled exits, such as inside an EVALUATE/error-handling paragraph that intentionally falls through.

Q17. Should you use nested IF statements or EVALUATE when validating multiple conditions?

Answer: EVALUATE is generally preferred over deeply nested IF/ELSE because it lays out all conditions in a flat, readable structure (similar to a SWITCH/CASE statement) and is far easier to validate, extend, and debug. Nested IFs become error-prone and difficult to read once more than two or three conditions are involved, and mismatched END-IF scope terminators are a common source of bugs.

```
EVALUATE TRUE
    WHEN WS-MARKS >= 90
        DISPLAY 'EXCELLENT'
    WHEN WS-MARKS >= 50
        DISPLAY 'PASS'
    WHEN OTHER
        DISPLAY 'FAIL'
END-EVALUATE.
```

Q18. Trace this EVALUATE TRUE logic: WHEN MARKS > 50 then WHEN MARKS = 60 — what is displayed when MARKS = 60?

Answer: It displays 'GOOD'. EVALUATE TRUE checks WHEN clauses strictly in the order they're written and stops at the first one that evaluates to TRUE — since 60 > 50 is true, the first WHEN fires and the second WHEN (MARKS = 60) is never reached. This is a common trap: condition ordering matters, and more specific conditions should generally be placed before broader ones.

Q19. Is a scope terminator (END-PERFORM) mandatory for an in-line PERFORM?

Answer: No — END-PERFORM is not syntactically required for an in-line PERFORM (the logic between PERFORM and the natural end of the paragraph can stand alone in some forms), but it is strongly recommended as good practice. Explicit scope terminators remove ambiguity about where a block ends, especially when nested with other IF/EVALUATE/PERFORM blocks, and most shops enforce it as a coding standard.

Q20. Can you define an array using OCCURS at the 01 level?

Answer: No. The OCCURS clause cannot be specified on a level-01 (record-level) item; it must be used on a subordinate (02–49) elementary or group item within a record. The 01 level defines the overall record structure, and the OCCURS table must live inside it.

```
* INCORRECT – OCCURS not allowed directly on 01
01 WS-STD-DETAILS OCCURS 50 TIMES.
   05 WS-STD-NAME PIC X(20).

* CORRECT
01 WS-STD-TABLE.
   05 WS-STD-DETAILS OCCURS 50 TIMES.
      10 WS-STD-NAME   PIC X(20).
      10 WS-STD-NUMBER PIC 9(05).
```

Q21. What happens if a program tries to access a table subscript/index beyond its OCCURS limit?

Answer: By default COBOL does not automatically check array bounds at runtime, so referencing an out-of-range subscript causes the program to read or overwrite adjacent storage — typically leading to unpredictable data corruption or an S0C4 (protection exception) abend. Compiling with the SSRANGE (subscript range checking) option forces the runtime to validate every subscript/index reference and raise a controlled error instead of silently corrupting memory; this is commonly enabled in test regions and disabled in production for performance.

Q22. What is the difference between a subscript and an index, and which performs better?

Answer: A subscript is an ordinary numeric WORKING-STORAGE variable (e.g., PIC 9(02)) that you declare, initialize, and increment yourself (ADD 1 TO WS-SUB); the compiler recalculates the byte displacement into the table from scratch on every reference, which is comparatively slower. An index is declared with the INDEXED BY clause on the OCCURS itself; it is maintained directly by the compiler as a displacement value, can only be changed via SET (not MOVE/ADD), and is faster because the compiler can use the pre-computed displacement directly rather than recalculating an offset. Indexes are generally preferred for performance-critical table processing; subscripts are simpler to reason about and more flexible for arithmetic.

```
01 WS-STD-TABLE.
   05 WS-STD-DETAILS OCCURS 3 TIMES INDEXED BY WS-IDX.
      10 WS-STD-NAME PIC X(20).

SET WS-IDX TO 2.
DISPLAY WS-STD-NAME(WS-IDX).
```

Q23. Given a 3-element table where the first occurrence stores 'SUSHMITA' and the second stores 'PRIYA', what are the subscript and index values for PRIYA?

Answer: As a subscript, PRIYA's position is simply its ordinal occurrence number — subscript value 2 (the second element). As an index, the value the compiler actually stores is not the occurrence number but the calculated byte displacement from the start of the table to that occurrence (displacement = (occurrence - 1) × element length), so the 'index value' is an internal address offset rather than a simple ordinal — it is not something application code reads or sets directly except via SET.

Q24. What typically causes an S0C7 abend?

Answer: S0C7 is a data exception, almost always caused by performing arithmetic or a numeric comparison on a numeric field that contains invalid (non-numeric) data — most commonly an uninitialized numeric WORKING-STORAGE field (one without a VALUE clause that was never MOVEd into before use) or a field corrupted by mismatched record layouts. The fix is to ensure every numeric field is properly initialized (VALUE ZERO or INITIALIZE) and that incoming file/table data is validated (e.g., with a NUMERIC test) before being used in arithmetic.

Q25. What is the difference between a static call and a dynamic call to a subprogram?

Answer: A static call (CALL 'PROGRAM-NAME' — a literal) is resolved at link-edit time: the called program's object code is bound directly into the calling program's load module, so it loads faster but any change to the subprogram requires re-link-editing the caller. A dynamic call (CALL WS-PROGRAM-NAME — a

variable, or CALL using IDENTIFICATION DIVISION DYNAMIC) is resolved at run time by the operating system loading the subprogram's separate load module on demand; this gives more flexibility (the subprogram can be changed/redeployed independently) at the cost of a small runtime loader overhead.

Q26. How do you pass data from one program (PGM-1) to another (PGM-2)?

Answer: Use the CALL … USING statement with matching WORKING-STORAGE variables in the calling program and corresponding LINKAGE SECTION variables (identical data type and length) in the called program, declared in the PROCEDURE DIVISION USING clause of the subprogram in the same order.

```
* Main program
WORKING-STORAGE SECTION.
01 WS-EMP-ID   PIC 9(05).
01 WS-EMP-NAME PIC X(30).
PROCEDURE DIVISION.
    CALL 'SUBPGM2' USING WS-EMP-ID, WS-EMP-NAME.

* Sub-program SUBPGM2
LINKAGE SECTION.
01 LS-EMP-ID   PIC 9(05).
01 LS-EMP-NAME PIC X(30).
PROCEDURE DIVISION USING LS-EMP-ID, LS-EMP-NAME.
    DISPLAY 'RECEIVED: ' LS-EMP-ID LS-EMP-NAME.
    GOBACK.
```

Q27. How do you know how much data can be transferred between two programs in a CALL?

Answer: The amount of data passed is determined entirely by the LINKAGE SECTION definitions in the called program — each parameter in the USING clause must match the calling program's corresponding WORKING-STORAGE field in data type and length (or be defined large enough to receive it), since COBOL passes parameters by reference (by address) by default, meaning the called program directly addresses the caller's storage. There is no single fixed size limit from COBOL itself; the limit is governed by how each LINKAGE SECTION field is declared and by the called program's region/storage constraints.

3. COBOL — Copybooks & Reusability

Q28. What is a copybook, and why is it used?

Answer: A copybook is a separate, reusable member containing standard data definitions (or sometimes procedural code) that is pulled into a COBOL program at compile time using the COPY statement. Copybooks ensure that multiple programs sharing the same file or table layout stay perfectly in sync, since a layout change only has to be made once in the copybook rather than in every program that uses it.

Q29. The same copybook needs to be used for both an input record and an output record in the same program — how do you avoid a duplicate-name compile error?

Answer: Use COPY … REPLACING to substitute the copybook's generic group-level name with a unique name for each use, so the same physical copybook can be included twice in one program without a duplicate data-name conflict.

```
FILE SECTION.
FD INPUT-FILE.
COPY CUSTOMER-REC
```

```
   REPLACING ==RECORD-REPLACE== BY ==INPUT-REC==.
FD OUTPUT-FILE.
COPY CUSTOMER-REC
   REPLACING ==RECORD-REPLACE== BY ==OUTPUT-REC==.
```

Q30. You need to add a new 2-character field but are told not to modify the physical copybook — how do you accommodate it?

Answer: Re-use REPLACING in the COPY statement at compile time to substitute an existing, unused copybook-level name with a new variable name suited to the new field, without physically editing the shared copybook member. This is typically done when the copybook is shared/locked across many programs and a structural change would require a much larger, riskier impact analysis.

```
COPY EMPLOYEE-DETAILS
   REPLACING ==EMP-DET1== BY ==EMP-DET2==.
```

Q31. A copybook field is used across many COBOL programs — how do you analyze the impact of changing it before making the change?

Answer: Use an impact-analysis / cross-reference tool such as Endevor's AUTOGEN, File-Aid, or a source-scan utility to search for every program that includes the copybook and references that specific field. For each impacted program, walk through how the field is used (MOVE, arithmetic, condition, output), then plan the remediation: update the copybook, recompile every affected COBOL program, adjust any JCL/control cards if record length or field position changed, and finally run unit and integration testing before promoting to production to ensure stability.

4. JCL — Job Control Language Concepts

Q32. If STEPLIB is defined in Step-2 and JOBLIB is defined on the job card, which library does each step use?

Answer: JOBLIB applies at the job level and is used by every step unless that step overrides it. STEPLIB, defined within a specific step, takes priority over JOBLIB for that step only. So Step-1 (no STEPLIB) uses JOBLIB, and Step-2 uses its own STEPLIB instead of JOBLIB — STEPLIB always overrides JOBLIB for the step in which it's coded.

Q33. Name common DISP (disposition) parameters and what they control.

Answer: DISP has up to three positional sub-parameters: the current status of the dataset (NEW, OLD, SHR, MOD), the action on normal step completion, and the action on abnormal termination — e.g., DISP=(NEW,CATLG,DELETE) means create a new dataset, catalog it if the step ends normally, and delete it if the step abends.

Q34. A user wants to append data to an existing file rather than overwrite it — what DISP value should be used?

Answer: Use DISP=MOD, which positions to the end of an existing dataset and writes new records after the existing content, instead of DISP=OLD/NEW which would overwrite from the beginning.

Q35. What is DISP=PASS used for?

Answer: DISP=PASS marks a dataset as 'in use, pass control to the next step within the same job' without releasing or catalog/uncatalog actions being taken yet — it defers the final disposition decision to a later step in the same job, useful for temporary work datasets shared across multiple steps.

Q36. What is the difference between a PROC and an EXEC statement?

Answer: An EXEC statement invokes a single program or a procedure for execution within a job step (EXEC PGM=... or EXEC PROC-NAME). A PROC is a reusable block of one or more pre-coded JCL steps — essentially a JCL subroutine — that can be invoked by name from multiple jobs, reducing duplication. Procedures are either instream (coded directly inside the JCL between PROC/PEND) or cataloged (stored as a separate member in a procedure library, the more common approach in production).

Q37. Regardless of whether the prior step abends or completes normally, you want the next step to always execute — what COND value achieves this?

Answer: COND=EVEN causes the step to execute even if a prior step abended. (By contrast, COND=ONLY executes the step only if a prior step abended/failed; without either, by default a step is bypassed if any prior step in the job abended.) For more complex multi-condition logic, IF/THEN/ELSE/ENDIF blocks in JCL provide finer control than COND.

Q38. How do you restart a job from a specific step inside a particular procedure invocation?

Answer: Use the RESTART parameter on the JOB statement, qualified with the step name within the proc invocation, in the form RESTART=stepname.procstepname — e.g., if JS0020 invokes PROC2 and you need to resume from STEP10 inside that invocation, code RESTART=JS0020.STEP10.

```
//JS0010  EXEC PROC1
//JS0020  EXEC PROC2

Restart from STEP10 within PROC2's invocation:
RESTART=JS0020.STEP10
```

Q39. How do you pass data from one JCL job to another dynamically?

Answer: Since separate JCL jobs are independent units of work, data is usually passed via persistent datasets (a job writes a dataset that the next job reads) or control cards/parameter files. In an orchestrated production environment, job schedulers such as Control-M or CA-7 are typically used to pass parameters between jobs dynamically based on trigger conditions. Within a single job stream (same job, multiple steps), DISP=PASS with a temporary dataset is the more direct mechanism.

Q40. How do you pass data from one JCL step to another step within the same job dynamically?

Answer: Use a temporary dataset (a name prefixed with &&, e.g., DSN=&&TEMP) that is created with DISP=(NEW,PASS) in an earlier step and referenced with DISP=(OLD,...) in a later step. The limitation is that temporary datasets only exist for the duration of the job and are automatically deleted once the job ends, so they cannot be used to pass data across separate jobs.

Q41. What happens when both STEPLIB and JOBLIB are coded in the same JCL?

Answer: STEPLIB, when present in a step, overrides JOBLIB for that step's program/load-module search — the system searches the STEPLIB concatenation first for that step. JOBLIB only applies to steps that do not have their own STEPLIB defined.

Q42. How do you pass parameters from one JCL to another, and what parameters are valid on an EXEC statement?

Answer: Within a step, the PARM parameter on the EXEC statement passes a literal string directly into the program's WORKING-STORAGE (typically picked up via a LINKAGE SECTION parameter or JCL-PARM area). Common EXEC statement parameters include PGM (or PROC), PARM, COND, REGION, TIME, and ACCT. Across separate JCLs, parameters are usually passed via a shared dataset, a control/parameter file read at program start, or a scheduler's variable-passing mechanism (e.g., Control-M).

Q43. How do you run multiple jobs in sequence?

Answer: Use a job scheduler (e.g., Control-M, CA-7, or simple JCL trigger files) to define job dependencies so that Job B only starts once Job A completes successfully; alternatively, chain steps within a single JCL job using COND/IF-THEN logic, or submit jobs from within a step using a utility that issues TSO SUBMIT for the next job.

Q44. Why use the INCLUDE statement in JCL?

Answer: INCLUDE pulls in a member containing pre-defined JCL statements (similar to a copybook for JCL) so common DD statements or step definitions can be reused across multiple jobs without re-typing them, keeping JCL consistent and easier to maintain.

Q45. A file has positions 20-22 containing a country code; how do you extract only the records where that code is 'US'?

Answer: Use DFSORT/SyncSORT with an INCLUDE COND that tests the specific byte range for the value 'US', so only matching records are written to the output dataset.

```
//SYSIN DD *
  INCLUDE COND=(20,3,CH,EQ,C'US')
/*
```

Q46. How do you concatenate multiple datasets in JCL?

Answer: List multiple DD statements under the same DDNAME without repeating the DDNAME — code the first dataset on the DD statement with the DDNAME, then list each additional dataset on its own DD statement (without a DDNAME) immediately following; all datasets must have compatible record formats and lengths (LRECL/RECFM).

```
//INFILE   DD DSN=PROD.FILE1,DISP=SHR
//         DD DSN=PROD.FILE2,DISP=SHR
//         DD DSN=PROD.FILE3,DISP=SHR
```

Q47. You need to feed 100 separate files as input to one COBOL program — how do you do this without coding 100 DD statements in the program?

Answer: Concatenate all 100 datasets under a single DD/DDNAME (subject to the JCL concatenation limits and matching record attributes), so the COBOL program simply reads from one logical file that spans all 100 physical datasets, or alternatively generate the DD concatenation list dynamically using a GDG (Generation Data Group) base if the files follow a generational naming pattern, referencing a relative GDG range (e.g., (0,-99)) instead of listing each one explicitly.

Q48. How do you remove duplicate records from a file, while also copying the duplicates into a separate file?

Answer: Use DFSORT with SORT FIELDS to establish key order, then use SUM FIELDS=NONE to eliminate exact duplicates in the primary output, combined with an OUTFIL statement using the SAVE option (or an ICETOOL SELECT operator) to write the discarded duplicate records to a second output dataset.

```
//SYSIN DD *
  SORT FIELDS=(1,10,CH,A)
  SUM FIELDS=NONE
  OUTFIL FNAMES=DUPSOUT,SAVE
/*
```

Q49. How can you control or limit the risk of a runaway/infinite loop in a COBOL program executing under JCL?

Answer: JCL cannot fix a logic defect, but it can bound the damage: the TIME parameter on the EXEC or JOB statement forces the system to abend the step once a specified CPU time threshold is exceeded, and the REGION parameter limits the amount of memory available, causing an S878/S80A abend if a runaway loop also leaks memory. Together these act as safety nets that stop a stuck job from consuming system resources indefinitely while engineers fix the underlying COBOL logic.

Q50. A 10-step JCL job abends at step 7 — what happens to steps 8, 9, and 10?

Answer: Steps 1 through 6 complete with condition code 0 as normal. Step 7 abends. By default, all subsequent steps (8, 9, 10) are bypassed and flushed (shown as 'FLUSH' in the job log) because the system assumes downstream steps depend on the failed step's output, unless those later steps explicitly carry a COND parameter (such as COND=EVEN) that allows them to run regardless of prior abends.

Q51. How do you override one specific step's DD statement when calling a cataloged procedure?

Answer: Qualify the override with the procedure step name and DD name in the calling JCL, in the form //procstepname.ddname DD DSN=new-dataset,DISP=SHR — this replaces just that DD's dataset for that one step of the proc invocation, without altering the cataloged procedure itself.

```
//STEP1.PROCSTEP1.OUTDD DD DSN=PROD.NEWFILE,DISP=SHR
```

Q52. What is the purpose of a control card in JCL/sort processing?

Answer: A control card supplies runtime parameters or criteria to a utility program (most commonly via SYSIN) — for example, SORT FIELDS=COPY to pass records through unchanged, INCLUDE/OMIT conditions

to filter records, or DSN overrides — without hard-coding that logic into the program itself, making batch jobs configurable and reusable.

Q53. What are common abend codes encountered in mainframe batch processing, and what causes them?

Answer: S0C4 — protection exception, typically an addressing error (e.g., subscript out of bounds, uninitialized pointer, mismatched copybook layout). S0C7 — data exception from invalid (non-numeric) data in a numeric arithmetic operation. S0C1 — operation exception, often from branching into bad/uninitialized code, frequently from a missing or corrupted load module. S322 — job exceeded its TIME parameter limit. S806/S813 — load module or dataset not found in the library concatenation. SQLCODE -803 — duplicate key insert. SQLCODE -805 — DBRM/plan mismatch, requiring a re-bind. SQLCODE -904 — resource unavailable (e.g., tablespace not started or locked). User abend U0100 and similar — application-defined abend codes raised explicitly by program logic (e.g., via CEE3ABD or ABEND statements) to signal specific business-rule failures.

5. GDG, SORT & Mainframe Utilities

Q54. What is a GDG (Generation Data Group), and how does generation numbering work?

Answer: A GDG is a base (catalog) entry that groups chronologically related datasets — generations — under one name, allowing programs to reference 'the current version', 'the prior version', etc., without hard-coding a specific dataset name. A GDG base is typically created via IDCAMS DEFINE GDG. New generations are referenced relative to the base using a signed integer suffix: +1 refers to the next generation to be created in the current job (used when creating a new generation), 0 refers to the most recently completed/cataloged generation (used when reading the latest version as input), and negative numbers (-1, -2, ...) refer to progressively older generations.

Q55. In a 2-step job — Step 1 creates GDG(+1), Step 2 reads it as input — what generation number does Step 2 use, and what happens on restart?

Answer: Step 2 reads the generation Step 1 just created using relative generation 0 (the most recently cataloged generation), not +1 — +1 is only valid for creating a brand-new generation, not reading one. If the job abends and is restarted after Step 1 has already completed and cataloged the new generation, that generation is now treated as generation 0 for the restart, so Step 2 continues to correctly reference it as 0.

Q56. How do you delete an entire GDG (base and all generations)?

Answer: Use IDCAMS with the DELETE command and the FORCE option, which removes the GDG base along with all of its generation datasets in one operation (without FORCE, you would need to delete every generation individually before deleting the base).

```
//SYSIN DD *
  DELETE GDGNAME FORCE
/*
```

Q57. How do you split a file so the first 10 records go to one output file and records 30-40 go to another?

Answer: Use DFSORT with multiple OUTFIL statements, each using the STARTREC/ENDREC (or INCLUDE-by-record-number via an INREC sequence number) to define the desired record range for each output dataset.

```
//SYSIN DD *
  SORT FIELDS=COPY
  OUTFIL FNAMES=OUT1,STARTREC=1,ENDREC=10
  OUTFIL FNAMES=OUT2,STARTREC=30,ENDREC=40
/*
```

Q58. Two input files share a 5-byte key in positions 1-5; how do you produce both matched (paired) and unmatched (unpaired) records?

Answer: Use DFSORT's JOINKEYS facility to join the two files on the common key, then use OUTFIL with INCLUDE conditions on a join-indicator field to separate the output into paired and unpaired record sets. Equivalently, this can be coded natively in COBOL by reading both files in key sequence and comparing keys record by record, writing to the paired or unpaired output file depending on whether the keys match, and advancing whichever file has the lower key.

```
//SYSIN    DD *
  JOINKEYS FILE=F1,FIELDS=(1,5,A)
  JOINKEYS FILE=F2,FIELDS=(1,5,A)
  JOIN UNPAIRED,F1,F2
  REFORMAT FIELDS=(F1:1,50,F2:1,50)
  SORT FIELDS=COPY
  OUTFIL FNAMES=SORTOUT,INCLUDE=(51,5,CH,EQ,1,5,CH)
  OUTFIL FNAMES=UNPAIRED,INCLUDE=(51,5,CH,NE,1,5,CH)

COBOL match/merge logic:
  Read File1 and File2 records.
  Compare the 5-byte keys.
    - Keys equal      -> write to Paired output, read next from both files.
    - File1 key lower -> write File1 record to Unpaired, read next from File1.
    - File2 key lower -> write File2 record to Unpaired, read next from File2.
  Repeat until both files reach end-of-file.
```

Q59. A fixed-format file has a 15-byte first column; how do you replace a value (e.g., 'SRI') located in bytes 13-15 with another value ('KMR')?

Answer: Use DFSORT's OUTREC FINDREP function, which scans for a literal value anywhere within a defined field and replaces it, without needing to know the exact starting offset of the value within that range.

```
//SYSIN DD *
  OUTREC FINDREP=(IN=C'SRI',OUT=C'KMR')
/*
```

Q60. An address field contains both a street name and a pin code; how do you update only the pin code portion?

Answer: Use reference modification or substring functions to isolate just the pin-code sub-field within the larger address field and update that portion in place, leaving the street-name portion untouched — in SQL contexts the equivalent is the SUBSTR function combined with CONCAT to rebuild the field with the new pin code spliced in.

```
* COBOL reference modification
MOVE NEW-PINCODE TO WS-ADDRESS(46:6).

-- SQL equivalent
UPDATE CUSTOMER
SET ADDRESS = CONCAT(SUBSTR(ADDRESS,1,45), :NEW-PINCODE)
WHERE CUST-ID = :WS-CUST-ID;
```

6. DB2 / SQL

Q61. What is SQLCA, and what are its key components?

Answer: SQLCA (SQL Communication Area) is a standard structure automatically populated by DB2 after every EXEC SQL statement, used to report the outcome of that statement. Its key fields include SQLCODE (0 = success, negative = error, positive = warning such as +100 for 'no rows found'), SQLSTATE (a standardized 5-character status code), and SQLERRD — an array where SQLERRD(3) specifically returns the number of rows affected by the last INSERT/UPDATE/DELETE. Programs include it via INCLUDE SQLCA and check it after every SQL statement to drive error handling.

Q62. After an UPDATE statement, how do you find out how many rows were actually updated?

Answer: Check SQLERRD(3) within the SQLCA immediately after the UPDATE executes — DB2 places the count of rows affected by the last INSERT, UPDATE, or DELETE into that field automatically.

```
EXEC SQL
   UPDATE EMPLOYEE
   SET SALARY = SALARY + 1000
   WHERE DEPT = 'IT'
END-EXEC.

IF SQLCODE = 0
   DISPLAY 'ROWS UPDATED: ' SQLERRD(3)
ELSE
   DISPLAY 'ERROR: ' SQLCODE
END-IF.
```

Q63. A cursor's SELECT/FETCH is returning 2 rows when only 1 row is needed — how do you restrict it to one row?

Answer: Add the FETCH FIRST n ROWS ONLY clause to the cursor's SELECT statement, which caps the result set to the specified number of rows directly at the database level, rather than fetching extra rows and discarding them in application logic.

```
EXEC SQL
DECLARE C1 CURSOR FOR
SELECT EMP_NAME
FROM EMPLOYEE
WHERE DEPT = 'IT'
FETCH FIRST 1 ROW ONLY
END-EXEC.
```

Q64. Can multiple cursors be open at the same time?

Answer: Yes. DB2 allows multiple cursors to be open concurrently within the same unit of work, as long as each is declared with a unique cursor name; this is commonly used when a program needs to cross-reference data from two different result sets simultaneously, such as comparing rows from two related queries.

Q65. Explain the typical cursor lifecycle in a COBOL-DB2 program.

Answer: A cursor is used to process a multi-row SQL result set one row at a time, since COBOL host variables can only hold a single row at once. The lifecycle is: DECLARE the cursor (associates a cursor name with a SELECT statement), OPEN the cursor (executes the query and positions before the first row), FETCH in a loop (retrieves one row per call into host variables, checking SQLCODE for +100 to detect end-of-data), and finally CLOSE the cursor to release the associated resources.

```
EXEC SQL DECLARE C1 CURSOR FOR
   SELECT EMP_ID, EMP_NAME FROM EMPLOYEE WHERE DEPT = 'IT'
END-EXEC.
EXEC SQL OPEN C1 END-EXEC.
PERFORM UNTIL SQLCODE = 100
   EXEC SQL FETCH C1 INTO :WS-EMP-ID, :WS-EMP-NAME END-EXEC
   IF SQLCODE = 0
      DISPLAY WS-EMP-ID WS-EMP-NAME
   END-IF
END-PERFORM.
EXEC SQL CLOSE C1 END-EXEC.
```

Q66. What does COMMIT do, and what is a 'cursor WITH HOLD'?

Answer: COMMIT ends the current unit of work, makes all changes since the last commit point permanent, and releases most DB2 locks held during that unit of work — it provides a checkpoint so that if a program later abends, restart logic only needs to reprocess work since the last commit rather than from the very beginning. By default, an ordinary cursor is automatically closed when a COMMIT is issued. Declaring a cursor WITH HOLD overrides this, keeping the cursor open and positioned across COMMIT points, which is essential for long-running batch programs that commit periodically (to release locks and limit log usage) while continuing to FETCH from the same cursor.

Q67. What do SQLCODEs -803, -805, -811, -502, -904, -911, and +100 mean, and how are they typically resolved?

Answer: -803: attempted to insert/update a row that would create a duplicate value in a column with a unique index or constraint — resolve by checking for the existing row before inserting, or using MERGE/UPDATE logic instead. -805: a DBRM-to-plan/package mismatch (the program's compiled DBRM doesn't match what's bound into the plan/package) — resolve by re-binding the package and/or plan after

recompiling. -811: a singleton SELECT INTO unexpectedly returned more than one row — resolve by adding a more selective WHERE clause, using FETCH FIRST 1 ROW ONLY, or converting to a cursor if multiple rows are legitimately expected. -502: attempted to OPEN a cursor that is already open — resolve by checking cursor state or ensuring CLOSE is called before re-opening. -904: a required resource (tablespace, index space, or database) is unavailable, often because it's stopped, in a pending state, or currently locked by another process — resolve by checking the object's status with -DISPLAY DATABASE and starting it, or waiting out/resolving the lock contention. -911: the unit of work was rolled back due to a deadlock or lock-timeout, and all locks were released — resolve by analyzing lock contention (often via DISPLAY/EXPLAIN), reducing the scope/duration of the unit of work, or adjusting commit frequency. +100: a positive, non-error warning meaning no rows were found for the SELECT/FETCH — this is the standard 'end of data' / 'not found' signal, not a failure.

Q68. How do you resolve SQLCODE -904 (resource unavailable, e.g., a tablespace that is locked)?

Answer: First identify which resource is unavailable and why, using -DISPLAY DATABASE(dbname) SPACENAM(*) to check its status (stopped, restricted, or held by another job/thread). If it's stopped, issue a -START DATABASE command; if it's locked by a long-running unit of work, identify and resolve that contention (e.g., wait for it to commit, or escalate to have it cancelled if it's a runaway process); reviewing commit frequency and isolation levels in the offending program often prevents recurrence.

Q69. What is referential integrity, and why does it matter?

Answer: Referential integrity is a database constraint that ensures relationships between related tables stay logically consistent — specifically, that every foreign key value in a child table corresponds to an existing primary key value in its parent table. This prevents 'orphan' records (e.g., an order row referencing a customer ID that doesn't exist) and is enforced by DB2 automatically once a foreign key relationship is defined, rejecting inserts/updates/deletes that would break the relationship.

Q70. What are DB2 isolation levels, and what do they control?

Answer: Isolation level determines how much a unit of work is shielded from concurrent changes made by other units of work, balancing data consistency against concurrency/performance. The standard DB2 levels are: Uncommitted Read (UR) — reads even uncommitted ('dirty') data from other transactions, fastest but least consistent; Cursor Stability (CS) — the default, only reads committed data and holds a lock on the current row only while positioned on it; Read Stability (RS) — holds locks on all rows that satisfied the query for the duration of the unit of work, preventing those specific rows from changing; Repeatable Read (RR) — the strictest, locks the entire range of qualifying rows, preventing both changes to existing rows and new rows from being inserted into the result set (preventing phantom reads), at the cost of higher lock contention.

Q71. What is a 'phantom read', and how does it occur?

Answer: A phantom read occurs when a transaction re-executes the same query twice within one unit of work and gets a different set of rows the second time, because another transaction inserted (or deleted) a qualifying row and committed in between. For example, Transaction A selects 10 rows where DEPT='IT'; Transaction B inserts a new IT employee and commits; Transaction A reruns the identical query and now sees 11 rows — the extra row is the 'phantom'. This is prevented by using the Repeatable Read (RR) isolation level, which locks the qualifying row range for the duration of the unit of work.

Q72. Can the MAX/MIN aggregate functions be used on an alphanumeric (character) column?

Answer: Yes — MAX and MIN work on character columns using lexicographic (character-by-character) comparison rather than numeric comparison. For example, MAX(EMP_NAME) over values 'ANITA', 'RAJ', 'ZARA' returns 'ZARA' because 'Z' is lexicographically the highest character, and MIN would return 'ANITA'. This becomes a common source of confusion with mixed alphanumeric strings like product codes (e.g., 'A12', 'B5', 'C100') — comparison is still purely character-based, so MAX might return 'C100' even though, numerically, 100 is not the largest value among 12, 5, and 100 when compared as strings starting with different leading characters; care must be taken to cast/pad such values consistently if true numeric ordering is required.

Q73. How do you eliminate duplicate values or duplicate rows in SQL?

Answer: Use SELECT DISTINCT to remove duplicate rows directly from a query's result set. To physically delete duplicate rows already stored in a table, typically pair ROW_NUMBER() (partitioned by the columns that define a 'duplicate') with a DELETE that removes all rows where the row number is greater than 1, keeping only the first occurrence of each duplicate group.

```
SELECT DISTINCT DEPT_NAME FROM EMPLOYEE;

-- Removing duplicate rows, keeping the first occurrence
DELETE FROM EMPLOYEE
WHERE EMP_ID IN (
  SELECT EMP_ID FROM (
    SELECT EMP_ID,
           ROW_NUMBER() OVER (PARTITION BY EMP_NAME, DEPT ORDER BY EMP_ID) AS
RN
    FROM EMPLOYEE
  ) X
  WHERE RN > 1
);
```

Q74. What is the difference between UNION and UNION ALL?

Answer: UNION combines the result sets of two or more SELECT statements and removes duplicate rows from the combined output, which requires an implicit sort/comparison step and is therefore slower. UNION ALL combines the result sets without removing duplicates, preserving every row from each query, and is faster since it skips the duplicate-elimination step — UNION ALL should be used whenever duplicates are known not to exist or are acceptable in the output.

Q75. How do you optimize a DB2 query that is exhibiting poor performance?

Answer: Start by running EXPLAIN to see the access path the optimizer chose (table space scan vs. index access, join method, sort operations) and look for missing or unused indexes on the columns used in WHERE/JOIN/ORDER BY clauses. Other common techniques include rewriting correlated subqueries as joins or using EXISTS more efficiently, ensuring statistics are current (RUNSTATS) so the optimizer has accurate cardinality information, avoiding functions on indexed columns in WHERE clauses (which prevent index usage), limiting the columns selected to only what's needed, and reviewing isolation level/locking strategy. For very large analytical/reporting queries, DB2 Accelerator (IDAA) can be used to offload complex analytical (OLAP) queries from the transactional DB2 engine to a dedicated high-speed analytics engine, improving response time without application changes.

Q76. What is a NULL indicator, and why is it needed in a COBOL-DB2 program?

Answer: A NULL indicator is a small signed binary host variable (typically PIC S9(4) COMP) paired with a regular host variable to represent SQL NULL, since COBOL data items have no native concept of 'no value' — an ordinary numeric or character field cannot itself represent NULL. When fetching a column that may contain NULL, an indicator variable is declared alongside the host variable; after the FETCH, a negative indicator value means the column was NULL (and the host variable's contents should be ignored), while zero or positive means the column had an actual value.

```
01 WS-SALARY        PIC S9(7)V99 COMP-3.
01 WS-SALARY-IND    PIC S9(4) COMP.

EXEC SQL
   SELECT SALARY INTO :WS-SALARY:WS-SALARY-IND
   FROM EMPLOYEE WHERE EMP_ID = :WS-EMP-ID
END-EXEC.

IF WS-SALARY-IND < 0
   DISPLAY 'SALARY IS NULL'
END-IF.
```

Q77. How do you compile and bind a COBOL+DB2 program?

Answer: First the source goes through a DB2 precompile step, which extracts every EXEC SQL statement into a separate DBRM (Database Request Module) and replaces it in the source with a call to the DB2 runtime, leaving pure COBOL behind. The COBOL side of the source is then compiled and link-edited normally into an executable load module. The DBRM side is processed by the BIND command, which binds it into a package (and the package is then bound into a plan, which may include multiple packages); the plan is what's referenced at execution time and contains the optimized access paths DB2 uses to run the embedded SQL.

Q78. Is a bind plan typically created separately for every program, and is the plan name the same across environments?

Answer: Plans are usually defined per package/application grouping rather than literally one plan per single program — a plan commonly bundles the packages of several related programs together. Plan names are not shared identically across environments; a common pattern is one plan per environment (e.g., a DEV plan, a TEST plan, and a separate PROD plan, often using a naming convention reflecting the region), so the same application logic is bound under environment-specific plan names as it's promoted through the lifecycle.

7. CICS & VSAM

Q79. What is DFHCOMMAREA in CICS?

Answer: DFHCOMMAREA (the Communication Area) is a CICS-provided storage area used to pass data between programs in a pseudo-conversational transaction — for example, carrying data from one screen interaction to the next, or from a calling program to a called program within the same CICS transaction. It's referenced in the LINKAGE SECTION of a CICS COBOL program and is conceptually similar to the LINKAGE SECTION used in a batch CALL...USING, but specifically scoped to CICS's request/response model across pseudo-conversational task switches.

Q80. What is VSAM, and why is it used in mainframe applications?

Answer: VSAM (Virtual Storage Access Method) is IBM's access method for organizing and managing data on direct-access storage, supporting several file organizations — KSDS (key-sequenced, indexed access), ESDS (entry-sequenced, sequential), and RRDS (relative-record, accessed by record number). It's widely used for high-performance, direct-access files (especially KSDS, which underlies most CICS online files) because it supports both random and sequential access efficiently, unlike a flat sequential file.

Q81. What is CICS, and what role does it play?

Answer: CICS (Customer Information Control System) is IBM's transaction processing monitor that manages online, real-time transactions on the mainframe — handling terminal/screen input, multi-user concurrency, task scheduling, and resource management (files, DB2, queues) so that individual application programs don't need to manage those concerns themselves. It is the runtime environment underneath most interactive (as opposed to batch) mainframe applications.

8. Mainframe Tools, File Access Modes & Production Support

Q82. How do you declare an indexed (VSAM KSDS) file in COBOL?

Answer: Declare it in the FILE-CONTROL paragraph with ORGANIZATION INDEXED, specify the RECORD KEY (the field within the record layout used as the unique key), and choose an ACCESS MODE appropriate to how the program will use the file.

```
FILE-CONTROL.
    SELECT CUSTOMER-FILE ASSIGN TO CUSTFILE
        ORGANIZATION IS INDEXED
        ACCESS MODE IS DYNAMIC
        RECORD KEY IS CUST-ID
        FILE STATUS IS WS-FILE-STATUS.
```

Q83. What are the different file ACCESS MODE options, what is the default, and what's the difference between RANDOM and DYNAMIC?

Answer: The three access modes are SEQUENTIAL (records processed strictly in physical/key order, one after another), RANDOM (a specific record is retrieved directly by its key on each I/O, with no implied ordering between reads), and DYNAMIC (a combination of both — the program can switch between sequential reads and random key-based reads in the same run). The default access mode, when none is specified, is SEQUENTIAL. The key difference between RANDOM and DYNAMIC is flexibility: RANDOM only supports direct-by-key access (READ with the key set, one record at a time), while DYNAMIC additionally allows START plus sequential READ NEXT processing from that point forward — useful when a program needs to locate a starting record by key and then browse forward sequentially from there.

Q84. A file needs to be both read and written within the same OPEN — what file mode should be used?

Answer: Use I-O mode (OPEN I-O), which allows both reading existing records and rewriting/updating them in place within a single open, as opposed to OPEN INPUT (read-only) or OPEN OUTPUT (write-only, which also resets the file).

Q85. What is an alternate key, and how/why is it used?

Answer: An alternate key (or alternate index, AIX, in VSAM terms) is a secondary key defined on an indexed file in addition to its primary RECORD KEY, allowing records to be retrieved by a different field — for example, retrieving a customer record by phone number in addition to by customer ID. In COBOL, it's declared with the ALTERNATE RECORD KEY clause in FILE-CONTROL, and the underlying VSAM file must have a corresponding alternate index built (via IDCAMS DEFINE AIX/BLDINDEX) over the base cluster.

```
SELECT CUSTOMER-FILE ASSIGN TO CUSTFILE
    ORGANIZATION IS INDEXED
    RECORD KEY IS CUST-ID
    ALTERNATE RECORD KEY IS CUST-PHONE WITH DUPLICATES.
```

Q86. What is the function of Endevor (or a similar source code management tool) in a mainframe environment?

Answer: Endevor is a source code/version control and software change management tool used to govern the lifecycle of mainframe programs as they move from development through to production. Typical workflow: a component (program, copybook, JCL) is added or checked out into the development region for changes; once development and unit testing are complete, the change is packaged; the package is submitted for approval; once approved, a team lead promotes the package to the next environment (e.g., test, then production). Endevor maintains full version history, supports retrieving prior production versions, packages related elements together (JCL, control cards, load modules) for a coordinated deployment, and provides an audit trail of who changed what and when.

Q87. What is a File Manager (File-Aid) tool used for?

Answer: A file/data utility such as File-Aid or File Manager is used to browse, edit, and validate data in a file using its associated copybook layout, rather than viewing raw, unformatted hex/character data — letting a developer see field-by-field, named values exactly as the program would interpret them. It's commonly used to inspect test data, manually correct specific field values for test scenarios, and compare files.

Q88. How do you find out which programs use a particular cataloged procedure?

Answer: Search the procedure library (PROCLIB) cross-reference, typically through ISPF option 3.4 (dataset list utility) by opening the relevant JCL/PROCLIB and using the member search command to find every JCL member that contains an EXEC referencing the proc name, or by using Endevor's component cross-reference / AUTOGEN-style search if the procedure is tracked under change management.

Q89. Where do you check for job execution errors after a batch job runs?

Answer: Review the job's SPOOL output, specifically the SYSOUT/SYSPRINT/JESMSGLG datasets accessible through SDSF (or an equivalent spool browsing tool) — the job log shows the condition code (return code) for each step, and SYSPRINT/SYSOUT contains program DISPLAY output and any abend or SQL error messages for troubleshooting.

Q90. What testing documentation is typically prepared for a mainframe change?

Answer: A Unit Test Plan (UTP), usually maintained in a spreadsheet, lists every test case with a description, the steps to execute it, the expected result, and the actual result; a Unit Test Report (UTR) then records the pass/fail outcome for each test case along with supporting evidence (screenshots, output

extracts, comparison reports). These artifacts demonstrate that a change was validated against its requirements before being handed off for QA/regression testing.

Q91. What is regression testing?

Answer: Regression testing verifies that existing, previously working functionality has not been unintentionally broken by a new change — it re-runs a baseline set of test cases covering unrelated or adjacent functionality to confirm the change's 'blast radius' was limited to its intended scope.

Q92. What common mainframe utilities are used in batch processing?

Answer: IEFBR14 — a utility that does nothing except return a zero condition code, commonly used purely for its DD-statement side effects, such as allocating, deleting, or cataloging a dataset without running any real program logic. IEBGENER — a simple sequential file copy utility, often used to copy a file or generate a simple report from SYSIN to SYSOUT. SORT (DFSORT/SyncSORT) — sorts, merges, copies, and reformats records, and supports INCLUDE/OMIT filtering, FINDREP, and JOINKEYS matching. IDCAMS — the VSAM Access Method Services utility, used to define, delete, list, and manage VSAM clusters, GDG bases, and catalogs. IKJEFT01 — a TSO terminal monitor program (TMP) that allows TSO commands to be executed in batch via JCL.

Q93. What is the COND parameter used for, beyond step-level bypass logic, and how is it set to skip a particular step?

Answer: COND on a JOB or EXEC statement defines a condition under which the step (or job) should be bypassed, based on the return code of a previous step — for example, COND=(4,LT) bypasses the step if any prior step's return code is less than 4 is false... more precisely it bypasses if the test is true (return code less than 4), giving fine control to skip non-essential steps (such as optional reporting steps) when an earlier step already signaled a specific condition via its return code.

Q94. What is the RESTART parameter used for at a step level?

Answer: RESTART tells the system to resume job execution from a specific named step rather than from the beginning, which is useful after a job abends partway through — once the root cause is fixed, the job can be resubmitted with RESTART=stepname so the already-completed earlier steps are skipped and processing picks up exactly where it failed.

Q95. How do you execute only one specific step out of many in a job, without running the others?

Answer: Use the IEFBR14-based 'EXEC PGM=IEFBR14' bypass trick on the unwanted steps' DD allocations is one workaround, but the standard mechanism is to use COND= bypass logic or, more directly, mainframe tools/utilities like IEBEDIT (sometimes referred to informally as 'IEFBREDIT' in shorthand) that let you select and submit just a named subset of steps from a larger JCL stream for execution, effectively extracting and running only that one step.

Q96. How do dev (RG) and test (TUP) regions typically differ, even though both are used for testing?

Answer: Different mainframe shops define multiple pre-production regions (commonly named things like RG for development/unit testing and TUP/TEST for a more formal QA/system testing stage) so that developer-level testing and formal QA testing don't interfere with each other. Each region typically points

to its own separate set of program/load libraries and DSN naming conventions, so a change can be validated incrementally — first in the looser developer region, then in a more controlled, production-like QA region — before being promoted further.

Q97. How do you transfer files between a mainframe system and a desktop/local system?

Answer: Use FTP (or a secure variant such as SFTP/FTPS) to transmit datasets between the mainframe and a local machine; in a TSO/ISPF environment this can also be initiated through a command-line/terminal session (often invoked via a designated command shortcut in the ISPF environment) to push or pull files as needed.

Q98. How is a requirement typically captured and worked in a mainframe development team?

Answer: Requirements are commonly tracked using an Agile methodology with tools such as JIRA, where work is broken down into user stories with acceptance criteria; developers pick up stories from a sprint backlog, implement and unit test the change, and move the story through workflow states (in progress, code review, ready for QA, done) as it progresses.

Q99. What steps are typically followed when adding a new program to an existing system?

Answer: After writing the COBOL source: (1) pre-compile — for a COBOL+DB2 program, this extracts the embedded SQL into a DBRM; (2) compile and link-edit the COBOL to produce a load module, and bind the DBRM to create/update the package and plan; (3) unit test the program, validating it against expected results before promoting it through change management (e.g., Endevor) to subsequent test and production regions.

9. Behavioral & Project Experience Questions

Q100. Describe an incident where something went wrong in production and you stepped in to resolve it.

Answer: A strong answer follows a structure of situation, action, and result: identify a concrete production issue (e.g., a critical batch update job that normally completes in 1-2 hours was still running past 3 hours, risking an SLA breach), describe the proactive monitoring or alert that surfaced it, the specific corrective action taken (e.g., escalating to seniors and business stakeholders, getting confirmation before removing the problematic transaction/plan causing the slowdown, and communicating with the business to arrange reprocessing), and the outcome (the batch cycle was brought back on track and the SLA was met). The key elements interviewers look for are timely escalation, getting proper sign-off before taking a corrective action with business impact, and a clear, measurable resolution.

Q101. How do you approach impact analysis when a shared copybook field needs to change, through to deployment?

Answer: First identify every COBOL program (and any related JCL or downstream consumer) that includes the affected copybook and actually references the field in question, using a cross-reference/search tool. Update the copybook (or use COPY REPLACING if the copybook itself can't be touched), recompile every impacted program, review and adjust any JCL or control cards if the change affects record length or field positions (e.g., LRECL, SORT field positions), then execute unit testing on each changed program followed by integration testing across the full batch/online flow to confirm production stability before promotion.

Q102. What kinds of enhancement work are typical on an existing COBOL/DB2 application?

Answer: Enhancement work typically involves adding incremental logic to existing programs rather than building from scratch — for example, introducing additional business-rule validation into an existing COBOL or COBOL+DB2 program, inserting new conditional paragraphs, or replacing specific blocks of existing logic with updated rules, all while preserving the surrounding program's behavior and ensuring backward compatibility with existing data and downstream consumers.

Q103. How would you describe your experience level across COBOL, JCL, and DB2 in an interview?

Answer: Give an honest, specific self-assessment grounded in concrete examples rather than just a number — describe the type and volume of work done in each area (e.g., how many programs written from scratch versus enhanced, whether work included both batch and online/CICS, and the SQL complexity handled), since interviewers use a self-rating mainly as a prompt to probe deeper with follow-up questions rather than as a standalone metric.